Program Editing in a Software Development Environment

(DRAFT)

Steven P . Reiss [1]
Department of Computer Science
Brown University
Providence, RI 02912
spr@cs.brown.edu
(401)-863-7641, fax: (401)-863-7657

Abstract

This paper describes an approach to program editing that is suitable for a modern software development environment. Program editing involves editing whole programs and not just files. Our approach uses the data integration facilities of the Desert environment to combine information about the overall system with that of the file being edited. It is built on top of the commercial word processing tool, FrameMaker , to provide high-quality presentations of both text and graphics. It uses an API to provide both insets supporting different tools for non-textual software artifacts (e.g user interfaces, class diagrams, and visualizations) and formatted program text. The API does minimal incremental parsing to enable the proper formatting and to relate the text being editing to the rest of the system being developed.

1.0 Goals for a Program Editor

The key components in a software development environment are the editors that allow the developer to modify the different software artifacts. Multiple editors are typically provided: graphical editors for modifying design diagrams, direct manipulation tools for building user interfaces, text editors for writing code, word processing systems for producing documentation, etc. Our goal in developing a new editor for a programming environment was to integrate these different editors as much as possible, providing a common framework to handle all software artifacts.

At the same time we wanted to provide enhanced services to the programmer through the editor . Our primary goal was to provide a true program editor . A program editor must be designed for editing programs, not files. This does not mean that it is a syntax-directed editor . Rather it implies that the editor must be aware that the file being edited is only one small part of a much larger system. The devel-

1. Support for this research was provided by the NSF under grants CCR9111507 and CCR9113226, by DARPA order 8225, by ONR grant N00014-91-J-4052, and by support from Sun Microsystems and NYNEX.

oper and the editor should be aware of dependencies between files, of references to definitions in other files and in libraries, etc. In short, the editor must be fully integrated into an overall programming environment.

A secondary goal was to provide high-quality display of program text. Although workstations have been used for programming for over twelve years, most program editing is still done in a simple, single-font, single color, text editor. Although studies have shown that using different presentation styles can enhance program readability and understandability [2], little has been done to take advantage of this for standard editing.

All this had to be done using an open system. We needed an editor that would be consistent with current software artifacts. It had to handle textual source files that were acceptable as input to the compiler; it had to handle the database files used by CASE tools such as Paradigm+ for OMT diagrams; it had to handle UIL files generated by user interface builders such as Builder Xcessory; and it had to handle word processing files for documentation, as well as natural language requirement and specification files. We also wanted an editor that could be used on existing programs and legacy code.

We are attempting to accomplish these goals by extending the commercial word processing system FrameMaker through the application program interface (API) it provides. Our tool uses the FrameMaker API to offer intelligent program editing with formatting and keystroke-based parsing and to integrate an external program database relating the file being edited to the rest of the program. It uses the inset capabilities of FrameMaker to offer access to other, specialized editors for different types of software artifacts. It uses the input filter mechanism of FrameMaker to allow different types of source files to be edited appropriately. An example can be seen in Figure 1. This shows both an inset (a simple OMT diagram of the object structure) and formatted C++ code. While our current implementation is heavily dependent on FrameMaker, the concepts and techniques we use can be applied to other editor frameworks or to the design of a complete editor.

2.0 Overview

FrameMaker provides a simple API consisting of four routines called using a remote procedure call interface. The routines service initialization requests, notifications, commands, and messages respectively. Of these, the primary work in our application is done using notifications. FrameMaker provides notification callbacks for most external events (i.e. open, close, save, and revert) and a single callback that is invoked after each command when control is being returned to the user. Our editor interface maintains the display and its internal structures using this latter callback.

One of the objectives of our editor interface was to provide well-formatted text. We wanted to utilize the formatting capabilities of a bit-mapped display to provide detailed information about the program and to assist the user in program creation

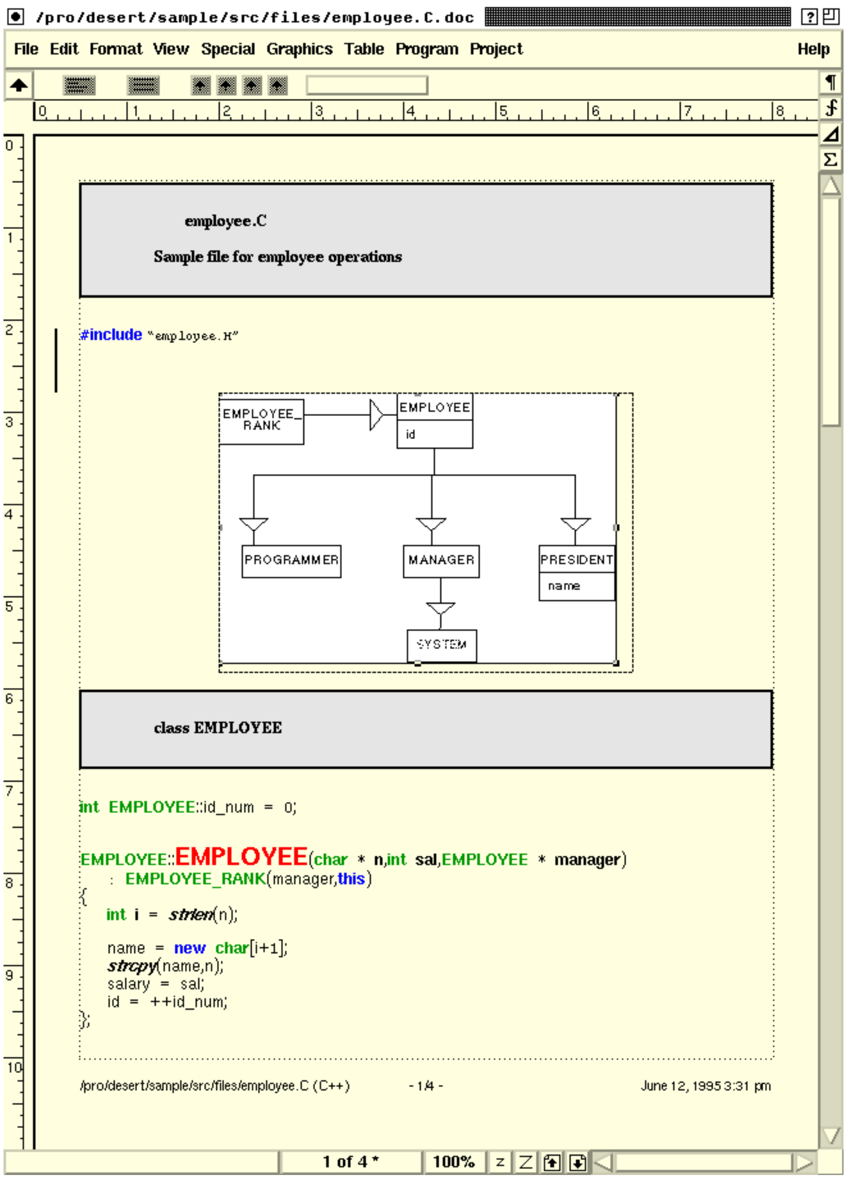


FIGURE 1. Sample FrameMaker session with inserts and formatting

and editing. This implied that we wanted to parse the resultant code as it was entered. While parsing on a keystroke basis is not new — it was done in the early 1980's in the COPE system at Cornell [1] — doing it outside of a syntax-directed editor , without direct control over the user 's input, in the context of a powerful word processing system, and in a language-independent manner is new .

Parsing presented several challenges. The first was obtaining the text to parse. While FrameMaker notified the API that a command was over , it did not provide any information about what the command did or what was changed. Our editor interface maintained its own version of the text being displayed. This was updated locally as much as possible using the current position before and after the command, with code to check for consistency and to reload everything if inconsistencies arose. By only considering the local neighborhoods of the previous and new position, we have been able to handle almost all commands with adequate performance. For those commands that we can not detect cleanly (e.g. a global find/change operation), we provide the user with a simple command to reload the text.

The second complexity that arose was determining how to represent the parse. We wanted to do incremental parsing to minimize the amount of work that was to be done on each keystroke. We wanted to handle incorrect programs since the editor will be used for initial program editing. We also wanted to handle a variety of different programming languages including some (such as C++) that are notoriously difficult to parse. The obvious choice here, parse trees, had several drawbacks. It was much more complex than necessary since we only needed enough information for formatting. Parse trees are suitable when the program is correct, but it is difficult to define a parsable grammar that handles incorrect programs in a clean way. They are also typically used to represent the language-portion of the program and ignore comments and white space which we needed to maintain and format properly. Parse trees also imply that full parsing must be done, and even incremental parsing can be complex and time-consuming. For example, adding a left brace in a C program would normally invalidate the remainder of the file and cause it to be reparsed.

We chose a simpler representation that meets our needs without the difficulties of parse trees. We represent the parse as two structures, a stream of tokens, generated on a line basis, and a symbol table. The token stream serves multiple purposes. It provides input for the simple parsing that is needed to maintain the symbol table and to find the proper indentation for a line. It also provides a basis for formatting using FrameMaker's paragraph and character formats.

In the next three sections we look at symbol table management, parsing, and formatting in more detail.

3.0 Symbol Table Management

Symbol table management in our editor interface is similar to that provided by a compiler in offering the capability to define and lookup scopes and names inside scopes. It differs in three aspects. First, it provides incremental facilities whereby symbols can be dynamically defined and undefined to support incremental parsing. Second, it provides the facilities to support incomplete programs by maintaining the implicit type of undefined symbols. Finally, it provides a connection with the system database to facilitate lookup and cross-referencing of names defined and used in other parts of the system.

The basic components of the symbol table are scopes and objects. Scopes have a scope type, a parent scope, an alternate parent, a set of super class scopes and a list of the symbols defined in the scope. The scope type determines how the scope behaves for lookup and definition, i.e. whether definitions are allowed, what types of names should be defined in this scope, etc. The parent scope reflects the actual scope nesting. The alternate parent is used when the scope is actually nested twice, for example a method body in C++ is nested in both the class scope and the file scope. Symbols contain their name, the symbol type, the scope of the definition, and the scope associated with the symbol (i.e. a class scope for a type name).

Scopes support the normal operations of defining names and looking them up. An undefine operation is provided to support incremental parsing. An assume operation is used to handle undefined symbols due to incomplete programs. The assume operation takes a name and the symbol type and works similar to define except that it only creates a name in the outermost scope regardless of what scope it is called on and it sets a flag in that name indicating that the name is assumed. If the name was already defined the assume operation is ignored. If the name had been previously assumed, then the symbol type of the previous definition and the symbol type of this definition are checked to determine a new symbol type for a name. This is needed to handle names that can be used ambiguously such as type names that can be used as functions (for casting or constructors) and functions that can be used as pointer -to-function variables.

The symbol table is responsible for most of the interconnections between the file being edited and the program database. This is managed through the outermost scope. This scope automatically establishes a connection with the system database. It determines the set of include files referenced by the file being edited and creates a lookup context in that database consisting of these files and the files they include. The first time a name is looked up in the global scope, a request is made to the system database to find this name in the lookup context and the information returned is used to define the symbol.

The global scope is also used for managing dynamic cross-reference links based on uses and definitions. When the editor interface needs either the set of definitions or the set of references for a name that is defined or could be accessed externally , it sends a corresponding request to the system database. External definitions are actually handled by caching the information returned by the initial symbol table lookup. External references (either based on use-def chains or a name pattern) are queried explicitly when needed.

4.0 Parsing

Parsing in the API is done for two purposes. The first is to maintain the symbol table and provide a local version of use-def chains to support implicit links within the file. Relating identifier uses to the appropriate definitions is one of the key functions of the system database and is crucial for program editing. The system database, however , is not accurate for a file that is in the process of being edited or created. The editor thus compromises, constructing use-def relations by consulting the database for items from files that are not being edited, and using the local parse structures for files that are being edited. The second objective is to allow the editor to format the text to make it more readable along the lines proposed in [2]. This

```
int fct(Bool x) {
   int a = 5;
   return x+a;
}
```

a) Sample Program

```
KEY_INT ID=fct LPR ID=Bool ID=x RPR
     LBR
KEY_INT ID=a EQ INT SEMI
KEY_RETURN ID=x OP ID=a SEMI
RBR
```

b) Original Token Stream

```
KEY_INT FCT_DEF=fct LPR_ARG1
     ID=Bool VAR_DEF=x RPR_ARG1 LBR
KEY_INT VAR_DEF=a EQ INT SEMI
KEY_RETURN ID=x OP ID=a
RBR
```

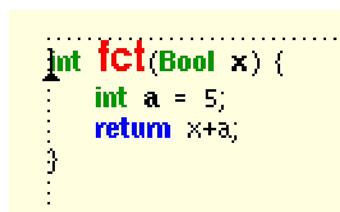
c) After Declaration Parsing

```
KEY_INT FCT_DEF=fct LPR_ARG1
     TYPE_REF=Bool VAR_DEF=x
     RPR_ARG1 LBR
KEY_INT VAR_DEF=a EQ INT SEMI
KEY_RETURN VAR_REF=x OP VAR_REF=a
RBR
```

d) Final Token Stream

```
Outer Scope
      Bool : Type (Assumed)
   File Scope
         fct : Function
      Function Scope
         Argument List Scope
               x : Variable
            Local Scope
                  a : Variable
```

e) Resultant Symbol Table



f) Resultant display

FIGURE 2. Parsing example

required token-based formatting and required the editor to distinguish between tokens representing different types of objects. It also required identifying and isolating block and in-line comments.

The simplest representation that offers these capabilities consists of a token stream and a symbol table. The parsing algorithm builds this incrementally, looking at lines as they change and updating the representation appropriately. It achieves language independence by using language-specific routines for the actual generation of an initial token stream for a line and the mapping of this token stream to a qualified token stream with the appropriate symbol table manipulations.

Parsing within the editor interface is done in three phases. The first phase consists of tokenizing the source. Figure 2a show a simple source example which is tokenized as shown in Figure 2b. Tokenization is done incrementally one line at a time. In order to handle complex tokens, the editor maintains a start and end state

Token	New Token	Situation
COMMA	COMMA_ENUM	Comma in an enumeration list
	COMMA_ARG	Comma in function argument list
COLON	COLON_CLASS	Colon separating class from superclass
	COLON_INIT	Colon starting initializer list for constructor
LPR	LPR_ARG0	Left parenthesis for external/static function
	LPR_ARG1	Left parenthesis for ANSI or C++ function definition
	LPR_ARG2	Left parenthesis for KR-style function definition
RPR	RPR_ARG0	Right parenthesis for external/static function
	RPR_ARG1	Right parenthesis for ANSI or C++ function definition
	RPR_ARG2	Right parenthesis for KR-style function definition
ID	TYPE_TAG_REF	Superclass name
	VAR_REF	Variable in KR-style argument list

TABLE 1. Non-definition tokens modifies during first parsing pass for C and C++

for each line. These indicate whether the line is a normal line, is in the middle of a comment, is a macro definition, or is some other type of preprocessor line. When a line changes, its previous tokens are discarded and new tokens are computed for it. If the end state for the line does not match the starting state for the subsequent line, new tokens will be computed for it and subsequent lines as needed.

Once tokens have been computed for a line, it can be parsed. The editor attempts to support incremental parsing at the line level. The parser is designed to scan a single declaration or statement at a time. To facilitate the combination of these, it keeps track of the starting and ending scope for each line. It also maintains two flags for each line. One indicates that this line is independent of the normal parsing rules (e.g. it is a preprocessor directive). The second indicates that parsing can be started with the first token on the line. When a line needs to be reparsed, the editor interface will find the appropriate starting point using these two flags and will start parsing at that point and continue until the line in question is parsed.

Generic parsing support is provided through a token stream object which allows the parser to scan tokens and to look ahead arbitrary amounts. It automatically notes when a line is being scanned and indicates that the parse for that line is being redone and the current line format might be invalid. It supports incremental parsing through a call from the parser that asks if the current token is at the end of a line. If it is not, the token stream tells the parser to continue parsing. If it is, the token stream sets the flag on the subsequent line indicating that parsing can start at that point and parsing stops.

The next parsing phase is responsible for identifying declarations and marking the items being declared by mapping the tokens of these items into appropriate definition tokens. In addition, it modifies other types of tokens to simplify the next parsing phase. The tokens modified are shown in Table 1. Declaration scanning is done with an extended FSA. The scanner builds a declaration structure that identifies the name and type tokens of the declaration as well as sets flags when certain

Token	Description
ENUM_MEMBER_DEF	Name inside enumeration list
TYPE_DEF	Type definition (from a typedef)
FIELD_DEF	Field definition
VAR_SDEF	Static variable definition
VAR_DEF	Local variable definition
METHOD_EDEF	External method definition
METHOD_DEF	Method definition with body
FUNCTION_EDEF	External function definition
FUNCTION_SDEF	Static function definition
FUNCTION_DEF	Function definition with body
VAR_EDEF	External variable definition
VAR_DEF	Variable definition
MACRO_DEF	Macro definition
ENUM_TAG_DEF	Enumeration type tag definition
TYPE_TAG_DEF	Struct, union, or class type tag definition
LABEL_DEF	Label definition

TABLE 2. Tokens set for declarations

keywords (i.e. static, extern, typedef) are detected. If no errors are detected, it uses the initial scope and the accumulated information to change the token type of the identifier being declared to one of the types shown in Table 2. The only symbol table manipulation that is done involves identifying tokens representing types and using the assume operator to define them as implicit types at the top level if they were not previously defined. An example of this can be seen in Figure 2c. Here the declaration scanner has identified three definitions (*fct*, *x*, and *a*) and replaced the corresponding ID tokens with FCT_DEF or VAR_DEF tokens. In addition, the parenthesis surrounding the argument list have been changed to LPR_ARG1 and RPR_ARG1 respectively to indicate an ANSI-style argument list, and the name *Bool* has been defined as an implicit type even though its token has not been changed.

The final parsing phase maintains the symbol table. This is done using a left-to-right scan through the source. Any token that starts or ends a scope causes the current scope to be updated. The name corresponding to any DEF token assigned on the first pass is entered into the symbol table in the current scope. Finally, any identifier token not modified by the first pass is looked up in the current scope. If it is defined, the token type is set accordingly. Otherwise, the local context is considered to determine if the token is implicitly a function, field, or method reference. If it is, then the token is defined as an assumed token of the given type. Otherwise, the token is either left as an ID token or changed to an UNDEF_ID token based on user preference. Figure 2d shows the result of this second scan on the previous examples. The ID tokens for the names *Bool*, *x*, and *a* have been changed to TYPE_REF and VAR_REF as appropriate.

Format	Description
F_Code	Normal line representing source code
F_BlankLine	Blank line that can serve as a separator
F_Preprocessor	Preprocessor line
F_CmmtLine	Line containing only a comment
F_CmmtStart	Blank line containing the anchor for a block comment
F_CmmtTable	Line inside a block comment

TABLE 3. Paragraph formats used by the editor interface

5.0 Formatting

The program text is formatted in two stages. The first involves assigning a paragraph format to the line based on the tokens of that line, while the second involves assigning a character format to each token based solely on the type of token. This simple approach is made possible by the parsing strategy which encodes the result of the parse in the token types.

This approach is quite flexible. Our objective was to approximate the work of Baecker and Marcus in designing an appropriate color display of the program text. Some changes had to be made to fit into the framework provided by FrameMaker and the notion of the user editing the text directly. Accommodations also had to be made so that existing source code could be read in and so that a readable ASCII file could be regenerated from the FrameMaker file.

We used six different paragraph formats to for formatting. These are shown in Table 3. Most lines are of type F_Code. More than one blank line in the middle of code or three or more blank lines immediately following a block comment are converted into paragraph type F_BlankLine. This is similar to F_Code except that it allows a page break while F_Code attempts to keep the current line with the subsequent line. The effect of this is to have page breaks occur at logical places in the program text and to provide the user with control over the breaks. The paragraph type F_CmmtLine is used for a single line containing only a comment. The type F_Preprocessor is used for all preprocessor lines.

The remaining paragraph types are used for formatting block comments. Taking the advice of Baecker and Marcus, we wanted comments to stand out by displaying them with an appropriate background. We accomplished this in FrameMaker by embedding the comment in a one-row, one-column table that is shaded appropriately. We provide four different types of comments which differ only in the color of the shading, gray for information, light red for warnings, light green for notices, and light yellow for alerts. The lines inside the comment have paragraph type F_CmmtTable, while the line containing the anchor for the table containing the comment is of type F_CmmtStart. The latter both allows a page break and is small enough to effectively be invisible.

Character Format	Description	Sample
Comment	In-line comment	*// comment*
Token	Single or multiple-character token	++
SymToken	Single or multiple-character token in Symbol font	*=
Keyword_Decl	Keyword as part of a declaration	**static**
Keyword_Stmt	Keyword starting a statement	**while**
Keywrod_Type	Keyword representing a type name	**float**
Keyword	Any other language keyword	**this**
String	String or character constant	"I'm a string"
Constant	Numeric constant	1.2345
Id	Identifier of unknown type	identifer
Id_undefined	Undefined identifier	undef_id

TABLE 4. Basic character formats

Identifier Type	Description	Sample Reference	Sample Definition
Constant	Enumeration constant or named constant	SAMPLE	**SAMPLE**
Function	Non-member function	***strcpy***	**strcpy**
Field	Class, structure, or union field	manages	**manages**
Label	Label as a goto target	label	**label**
Macro	Preprocessor macro	MACRO	MACRO
Method	Class member function	***method***	**method**
Type	Class, structure, union or enum tag; typedef name	EMPLOYEE	**EMPLOYEE**
Variable	Variable name	variable	**variable**

TABLE 5. Identifier types supported for formatting purposes

The key part of formatting is handled by the character types. Different character types are provided for different types of lexical units as shown in Table 4. In addition, a wide range of formats are provided for describing different types of identifiers. For each of the identifier types shown in Table 5, three formats exist, Id_<type>_ref to indicate a reference to an identifier of that type, Id_<type>_ext to indicate a reference to an external identifier of that type, and Id_<type>_def to indicate a definition of an identifier of the given type. Function declarations without a body (and similar method declarations) are viewed as external references rather than as definitions for formatting purposes. The result of this can be seen in Figure 3. Type names, both built-in and user-defined, are shown in dark green. Functions being defined have their name highlighted in red in a large font to stand out from the rest of the text. Functions being called are shown in bold italics. Names being declared, both in the argument list and in declarations, are emboldened. Keywords are display in blue. All other identifiers are shown in a standard font.

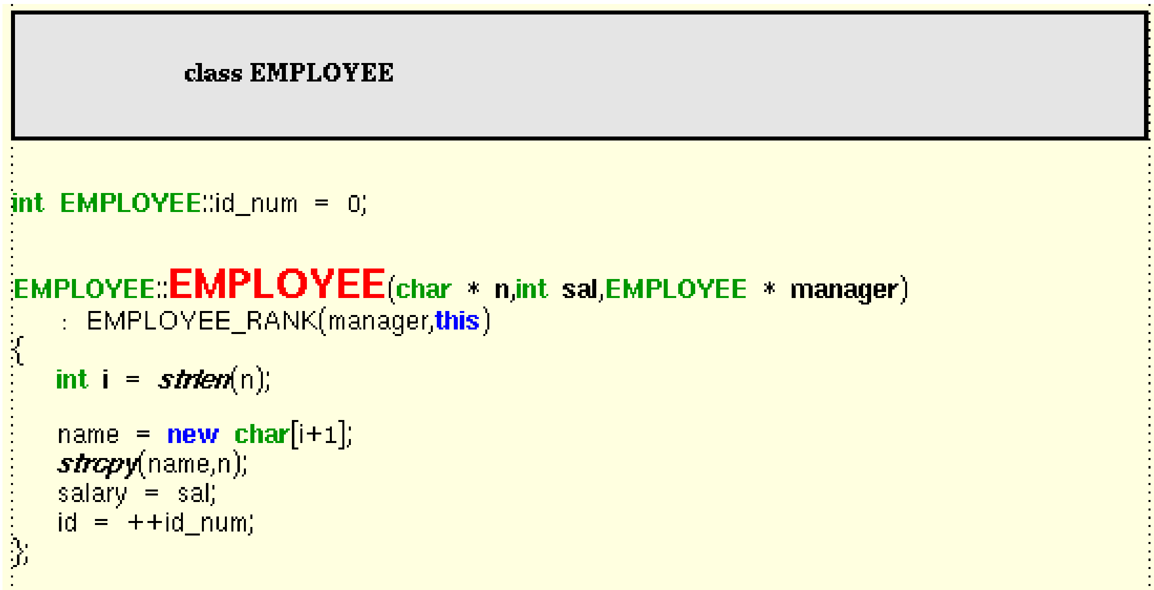


FIGURE 3. Example of formatting in the editor interface

6.0 Insets for Software Artifacts

One of the goals of an editor in a software development environment is to provide common access to the variety of software artifacts. Several of these artifacts, particularly the graphical ones, are built and maintained using special purpose editors: OMT diagrams describing an object-oriented design are maintained with an object design tool; user interface designs, recorded in UIL files, are edited using an interface builder . Our editor interface accommodates such tools using the inset mechanism provided by FrameMaker . The utility of insets or live-links goes beyond special purpose editors for software artifacts. We also use insets to provide visualizations of the user’s program as part of the source files. This is done by providing an inset-based interface to our program visualization tools [16].

Our editor interface provides a common front end for a variety of different inset editors. The front end manages two files for each interface. The first file is the input file to the tool, i.e. a file representing the software artifact. For Paradigm+ (and other tools that use a central database rather than individual files), we use the scripting extensions provided to generate a state file when the user saves the design. Our script to run Paradigm+ uses this state file to recreate the windows open at the time of the save. For Builder Xcessory , we use the UIL file that is saved by the tool. For our 3D visualization tool, we use the state file generated when the user saves the state of the diagram.

The second file managed by the front end is a file containing the image to display in the inset. This can either be a postscript file or a X11 image file. We have set up Paradigm+ to generate a postscript file when the state is saved (by editing the result of printing to a file). Similarly, our 3D visualization tool generates an X11 image file when the state is saved. Since we could not customize the interface builder tool we had, we instead wrote a small utility that takes a uil file and generates a corresponding X11 image. The common inset interface detects when the tool input file is older than the image file and, in this case, automatically runs the utility program to generate a new image file as needed. Integrating additional tools through the inset interface is simple and fast, taking less that half a day provided that the tool has a state file or one can be easily generated and there is a logical way of generating an image file to incorporate into FrameMaker.

7.0 Integration with Desert

The editor interface is designed to fit into our evolving programming environment, Desert. The editor needs to be able to take process requests from the environment. It also needs to use the environment both to understand the overall program and to allow the user to initiate commands in other tools from the editor.

Most of the editor's interaction is based on control integration and our experiences with the FIELD environment [15]. The editor defines a set of messages that can be sent to it by other tools to cause it to display a particular file and line. It defines messages that add a file-line pair to the current stack of gotos. This allows the editor to create a goto list consisting of all error messages generated by a compilation. The editor also sends out informative messages whenever a file is saved, opened, or closed. It also uses the message facility to provide commands to compile the current file or to build the system containing the current file.

Besides the use of control integration to facilitate the editor's interaction with the environment, the editor uses the common database provided by Desert to access information about names used in the current file but defined elsewhere in the system. The database, among other information, contains enough data to construct use-definition chains for each name in the system. The editor uses this to determine the proper type of any name that is not defined in the current file. It also uses these facilities to provide goto commands for the definitions or uses of a specified name, whether they appear in the current file or in files scattered throughout the system.

The other key part of the interaction of the editor with the underlying programming environment involves maintaining an open environment. While FrameMaker normally saves files in a binary format, such files are not acceptable input to other tools. In order to maintain an open environment, we had to insure that the editor could use the original source files. This is done in the interface in two parts. First, the interface provides a filter for Framemaker that takes an arbitrary source file and generates a FrameMaker file. This allows the user to open, from within FrameMaker, any source file and have it properly formatted and then displayed. Second, whenever the user requests that the current file be saved, the editor interface generates a copy of the modified source file and saves it as well. The interface attempts to make this file readable, mapping block comments in FrameMaker into *-encircled block comments in the source, while maintaining other indentation and formatting information. It also insures that the FrameMaker file can be rebuilt from the saved source file.

8.0 Related Work

There have been many editors written exclusively or primarily for programming. Many of these are syntax-directed editors, editors that parse the program and allow the programmer to work in terms of syntactic units of the underlying programming language. Syntax-directed editors were widely proposed and implemented in the early 1980's [4,6,7,14,18]. While some syntax-directed editors continue to be written, the general experience of those who wrote and used them was that users do not generally edit in terms of syntactic constructs and the syntax-directed features often get in the user's way. Programmer-knowledgable editors such as the programmer's apprentice have also been proposed [20]. These attempt to use artificial intelligence techniques to provide direction to the programmer. More recently, the Pan system attempts to use sophisticated semantic knowledge to provide programmer feedback [3]. Neither of these semantic approaches has been demonstrated as practical for realistically-sized programs.

More popular for programming are language-knowledgable editors such as emacs [10]. These editors know enough about the language being edited to do automatic indentation and simple error checking (such as parenthesis balancing). The primary example of such an editor is emacs. Extensions that have been included in such editors include the use of a tags file for handling simple links between a definition and use (assuming the name is unique), and, in the latest version of emacs, color highlighting based on regular expression patterns. Language-knowledgable editor can offer many of the features that our editor interface does However, they do not guarantee accuracy since the pattern matching (for indentation, links, and formatting) is all approximate. They are also not fully-functional word processing systems that are able to display and combine graphics with text. Editors for single-language systems, especially on PC's, also are language knowledgable. These provide the ability to automatically format the code, highlighting keywords, etc. as the user types. Our editor interface provides many of these capabilities in an open, multiple-language environment.

Other work that is related to ours includes work on literate programming [12]. This is a general attempt to use a single file for both documentation and code. A preprocessor extracts the code from the file when compilation is needed. Other processors can extract documentation, function headers, or other relevant information. Our editor interface can duplicate much of this. (For example, we currently only save lines with a paragraph type beginning with "F_" into the original source file.) Moreover, we are able to do this using the inherent capabilities of FrameMaker rather than extra formatting or punctuation characters added by the author to distinguish code and documentation. Our approach is closer to that used in the Cedar Mesa environment where the Cedar editor used the document structure and document tags to distinguish code and comments in the same document [19]. Our approach also provides these capabilities in an open environment, not depending on a single language or compilers that are aware of the document's structures.

Another area of related work involves the use of hypertext editors for programming. This is becoming more common with the advent of HTTP and editors built for the world-wide web [8]. These editors allow the user to create implicit or explicit links between the various parts of the program. Our approach can provide similar facilities, using the hypertext capabilities of FrameMaker for explicit links and the internal data structures and external database to provide implicit links. Multimedia insets are supported both by HTTP and by FrameMaker.

Another related topic is incremental parsing and semantic analysis. Incremental parsing typically shows how to extend standard parsing techniques to support incrementality [9]. A variety of techniques have been proposed for incremental semantic analysis, including attributed grammars [5], model-based functions [13],unification [17], and functions [11]. All of these techniques attempt to preserve full semantic information and depend on having the full program available and error-free. Our simplified techniques provide the information needed for editing without the cost of maintaining the more detailed information needed by a compiler. Moreover, they work well in the presence of syntactic and semantic errors.

9.0 Experiences

While the current version of the editor interface is still an early prototype, we have been using it to develop significant portions of the underlying Desert environment. Our experiences to date have shown that the concept of using a word processor such as FrameMaker for program editing is practical. Moreover, the additional formatting information and the feedback it provides is helpful during editing. We hope to do more extensive studies of the benefits and drawbacks of the editor interface once the interface and the environment that supports it is stable enough.

While the benefits of the editor in terms of the quality of the display, the additional information provided by formatting, the implicit links between uses and definitions, and language intelligent features such as automatic indentation are apparent when using the editor interface, there are drawbacks.

The first drawback is the performance of the editor interface. We have put a lot of effort into our editor interface to make common editing situations fast. Our experiences shows that the current setup has satisfactory performance (on a SparcStation 10) for normal editing. The performance issues arise when loading a new file, in saving a file to disk, and when reformatting the whole document is necessary. The problem here comes primarily from the need for extensive communication between our editor API and FrameMaker, with additional time required to access the external database. We are hoping that enhancements to interprocess communication and improvements to the FrameMaker API in the next version of FrameMaker will alleviate many of these problems. Delays caused by the interaction of the editor interface and the environment arise primarily when the underlying database system updates itself. We are beginning to work on minimizing these delays as well.

Other problems arise because our parsing methods are approximate. The primary drawback here is that fields and methods inside expressions are not resolved completely. This is alleviated somewhat by the use of unique method or field names or by naming conventions that differentiate field and method names. Our experience to date has indicated that this is probably not a serious problem since it is easy for the programmer to scan a list of references or definitions and pick and choose the ones that are actually relevant.

More general issues arise when the parser gets confused by some incorrect or unparsable construct. We have done our best to code around most of these and are correcting additional situations as they arise. The main problem left in this area involves understanding preprocessor macros. Our current implementation can detect the use of macros through the database system. However, it does not know the contents or meaning of a macro. The formatter, when a macro is encountered, does not attempt to set the type of names or do serious formatting inside a macro invocation. Because macros are used extensively in C and to some extent in C++ as well, we are attempting to develop a framework that would allow the user to characterize macro names for formatting, indentation, and symbol definition purposes. We hope that such a framework would handle most uses of macros that typically confuse our parsing approach without requiring that the parser actually find and expand macros.

Most of the remaining problems represent bugs or missing features in the current prototype implementation. For example, the current approach to determining what requires reformatting will miss some lines for some FrameMaker commands (such as global find/change). It also does not currently follow semantic links so that editing a definition does not automatically update all the uses. Similarly, the heuristics for handling ambiguous names (either from the database or from internal declarations) will sometimes associate the wrong type with a symbol. These problems only occur occasionally, and are easily handled by allowing the user to reformat (and hence reparse) the whole program or a selected portion at any time. The only drawback with this approach is that global reformatting currently involves considerable communications with FrameMaker and hence has performance problems.

In summary, we feel that the benefits to our approach to program editing, notably in offering program rather than file editing through connections to an underlying database and message-based interaction with the underlying environment, using a high-quality display, allowing the user to combine text and documentation in a single document, allowing interconnections between the program source and other software artifacts, and in offering all these capabilities in an open environment compatible with existing tools and programs, outweigh the current disadvantages. Moreover, we see that many of the current disadvantages can and will be overcome in the future and that this approach to program editing will be practical.

As such, we are continuing to work on the editor framework. We are first attempting to improve the performance, primarily through more sophisticated interaction with FrameMaker. We are working on adding additional languages and handling multiple languages in a single file. Yacc input files, for example, involve a language for defining the grammar and include both embedded and separate C or C++ code. We are working on support of conditional compilation using FrameMaker's conditional text facilities. We are working on support for multiple open files where the internal database is used for all open files rather than just the current one. We are working on better integration with the Desert environment including support for virtual files, files that are constructed by the environment by taking semantically related chunks from their different source files. Most importantly, we are attempting to build up a user community for the editor framework that will provide us with feedback and direction and a means for analyzing the utility and effectiveness of both high-quality text formatting and full-program editing.

10.0 Bibliography

1. James Archer, Jr. and Richard Conway, "COPE: a cooperative programming environment," Cornell TR81-459 (June 1981).

2. Ronald M. Baecker and Aaron Marcus, *Human Factors and Typography for More Readable Programs*, Addison-Wesley (1990).

3. Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter, "The Pan language-based editing system for integrated development environments," *ACM Software Engineering Notes* Vol. **15**(6) pp. 77-93 (December 1990).

4. Norman M. Delisle, David E. Menicosy, and Mayer D. Schwartz, "Viewing a programming environment as a single tool," *SIGPLAN Notices* Vol. **19**(5) pp. 49-56 (May 1984).

5. Alan Demers, Thomas Reps, and Tim Teitelbaum, "Incremental evaluation for attribute grammars with application to syntax-directed editors," Proc. 8th ACM Symposium on Principles of Programming Languages (1981).

6. Veronique Donzeau-Gouge, Gerard Heut, Gilles Kahn, and Bernard Lang, "Programming environments based on structured editors: the MENTOR Experience," in *Interactive Programming Environments*, ed. D. R. Barstow, H. E. Shrobe and E. Sandewall,McGraw-Hill, New York (1984).

7. Robert J. Ellison and Barbara J. Staudt, "The evolution of the GANDALF System," *Journal of Systems and Software* Vol. **5**(2)(May 1985).

8. James C. Ferrans, David W. Hurst, Michael A. Sennet, Burton M. Covnot, Wenguang Ji, Peter Kajka, and Wei Ouyang, "HyperWeb: a framework for hypermedia-based environments," *Software Engineering Notices* Vol. **17**(5) pp. 1-10 (December 1992).

9. Carlo Ghezzi and Dino Mandrioli, "Augmenting parsers to support incrementality," *JACM* Vol. **27**(3) pp. 564-579 (July 1980).

10. James Gosling, *Unix Emacs*, Carnegie-Mellon Computer Science Department (August 1982).

11. Gail E. Kaiser, “Semantics for Structure Editing Environments,” Ph.D. Dissertation, Carnegie-Mellon University (1985).

12. N. Ramsey, “Literate programming: weaving a language-independent WEB,” *CACM* Vol. **32**(9) pp. 1051-1055 (September 1989).

13. Steven P. Reiss, “An approach to incremental compilation,” *Proc. SIGPLAN '84 Symposium on Compiler Construction*, (June 1984).

14. Steven P. Reiss, “PECAN: program development systems that support multiple views,” *IEEE Trans. Soft. Eng.* Vol. **SE-11** pp. 276-284 (March 1985).

15. Steven P. Reiss, *FIELD: A Friendly Integrated Environment for Learning and Development*, Kluwer (1994).

16. Steven P. Reiss, “An engine for the 3D visualization of program information,” *Journal of Visual Languages*, (To appear in 1995).

17. Gregor Snelting and Wolfgang Henhapl, “Unification in many-sorted algebras as a device for incremental semantic analysis,” *Proc. 13th ACM POPL*, pp. 229-235 (January 1986).

18. Tim Teitelbaum and Thomas Reps, “The Cornell program synthesizer: a syntax-directed programming environment,” *CACM* Vol. **24**(9) pp. 563-573 (September 1981).

19. W. Teitelman, “A tour through Cedar,” *IEEE Software* Vol. **1**(2) pp. 44-73 (April 1984).

20. Richard C. Waters, “The programmer’s apprentice: knowledge based program editing,” in *Interactive Programming Environments*, ed. D. R. Barstow, H. E. Shrobe and E. Sandewall,McGraw-Hill, New York (1984).